

Seven Steps of Migrating a Program to a 64-bit System

[Andrey Karpov](#)

OOO "Program Verification Systems"

April 2009

Abstract

Introduction

1. The first step. 64-bit mode can be different. Let's sort it out

2. The second step. Find out if you need the 64-bit version of your product

[2.1. Applications' life-cycle duration](#)

[2.2. Resource-intensiveness of an application](#)

[2.3. Development of libraries](#)

[2.4. Dependence of your product on third-party libraries](#)

[2.5. Using 16-bit applications](#)

[2.6. Assembler code](#)

3. The third step. Toolkit

[3.1. A 64-bit compiler](#)

[3.2. 64-bit computers under control of 64-bit operation system](#)

[3.3. 64-bit versions of all the used libraries](#)

[3.4. Absence of embedded Assembler code](#)

[3.5. Testing methodology update](#)

[3.6. New data for testing](#)

[3.7. 64-bit security systems](#)

[3.8. Installer](#)

4. The fourth step. Setting of a project in Visual Studio 2005/2008

5. The fifth step. Compilation of an application

6. Diagnosis of hidden errors

[6.1. Explicit type conversion](#)

[6.2. Implicit type conversion](#)

[6.3. Bits and shifts](#)

[6.4. Magic numbers](#)

[6.5. Errors relating to using 32-bit variables as indexes](#)

[6.6. Errors relating to change of the types of the used functions](#)

[6.7. Diagnosis of hidden errors](#)

7. The seventh step. Update of the testing process

References

Abstract

The article describes the main steps which should be performed to correctly port 32-bit Windows applications on 64-bit Windows systems. Although the article is meant for developers using C/C++ in Visual Studio 2005/2008 environment, it will be also useful for other developers who plan to port their applications on 64-bit systems.

Introduction

The article describes the main problems facing developers who plan to port 32-bit programs on 64-bit systems. Of course the list of the issues considered is not complete, but we hope that we'll offer a more

detailed version of this article in future. The author would be glad to receive responses, comments and questions which will help increase information value of this article.

1. The first step. 64-bit mode can be different. Let's sort it out

Within the framework of a computer architecture by the term "[64-bit](#)" 64-bit integers and other 64-bit-sized data types are understood. By "64-bit" systems 64-bit microprocessor architectures (for example, EM64T, IA-64) or 64-bit operation system (for example, Windows XP Professional x64 Edition) can be understood [[1](#)].

AMD64 (or x86-64, Intel 64, EM64T, x64) is a 64-bit microprocessor architecture and a corresponding set of instructions developed by AMD company [[2](#)]. This set of instructions was licensed by Intel company under the name of EM64T (Intel64). AMD64 architecture is an extension of x86 architecture with full backward compatibility. The architecture became widespread as a basis of personal computers and workstations.

IA-64 is a 64-bit microprocessor architecture developed together by Intel and Hewlett Packard companies [[3](#)]. It is implemented in microprocessors Itanium and Itanium 2 [[4](#)]. The architecture is used mainly in multi-processor servers and cluster systems.

AMD64 and IA-64 are two different 64-bit architectures incompatible with each other. That's why developers have to decide at once if they need support of the both architectures or only one of them. In most cases, if you don't develop highly tailored software for cluster systems or don't implement your own high-performance DBMS, most likely you will have to implement support only of AMD64 architecture which is much more popular than IA-64. It especially concerns software for PC market which is nearly 100% occupied by AMD64 architecture.

Further in the article we'll speak only about AMD64 (EM64T, x64) architecture, as nowadays it is the most topical for application software developers.

Speaking about different architectures we should mention the notion "[Data model](#)". By a data model we understand correlations between type sizes accepted within the framework of the development environment. There can be several development tools sticking to different data types for one operation system. But usually only one model dominates which corresponds to the hardware and software environment most. Such an example is 64-bit Windows whose original data model is [LLP64](#). But for compatibility purposes 64-bit Windows supports execution of 32-bit programs which operate in [ILP32LL](#) data model mode. Table 1 gives information about the basic data models.

short	int	long	ptr	long long	Label	Examples
...	16	...	16	...	IP16	PDP-11 Unix (1973)
16	16	32	16	...	IP16L32	PDP-11 Unix (1977); multiple instructions for long
16	16	32	32	...	I16LP32	MC68000 (1982); Apply Macintosh 68K; Microsoft operation systems (plus extras for x86 segments)
16	32	32	32	...	ILP32	IBM 370; VAX Unix; many workstation
16	32	32	32	64	ILP32LL or ILP32LL64	Microsoft Win32; Amdahl; Convex; 1990 Unix systems; Like IP16L32, for same reason; multiple instructions for long long
16	32	32	64	64	LLP64 or IL32LLP64 or P64	64-bit systems Microsoft Win64 (X64 / IA64) Most Unix systems (Linux, Solaris, DEC OSF/1 Alpha, SGI Irix, HP UX 11) HAL; logical analog of ILP32 UNICOS
16	32	64	64	64	LP64 or I32LP64	
16	64	64	64	64	ILP64	
64	64	64	64	64	SILP64	

Table 1. Data models.

A data model being used influences the process of developing 64-bit applications as you need to keep in mind sizes of the data being used in programs' code [5].

2. The second step. Find out if you need the 64-bit version of your product

You should begin mastering 64-bit systems with the question: "Do I really need to rebuild my project for a 64-bit system?" You give an answer to this question but only after you have thought it over, without hurry. On the one hand, you can lag behind your rivals if you don't offer 64-bit solutions. On the other hand, you can just waste your time developing a 64-bit application which won't provide any competitive advantages.

Let's list the basic factors that will help you make up your mind.

2.1. Applications' life-cycle duration

You shouldn't create the 64-bit version of an application with a short life-cycle. Thanks to [WOW64](#) subsystem old 32-bit applications operate rather well on 64-bit Windows systems and that's why there is no sense in making a program 64-bit for it won't be supported in 2 years [6]. Moreover, practice shows that porting on 64-bit Windows versions has been delayed and perhaps most of your users will use only the 32-bit version of your program solution in near-term outlook.

If you plan long-term development and support of a program product, you should begin to work over the 64-bit version of your solution. You can do this without hurry but keep in mind that the longer you don't have a complete 64-bit version, the more difficulties you will face in supporting this application installed on 64-bit Windows versions.

2.2. Resource-intensiveness of an application

Recompilation of a program for a 64-bit system will allow it to use large sizes of main memory and will also speed up its operation by 5-15%. Increase in 5-10% will be gained due to using the 64-bit processor's architectural abilities, for example a larger number of registers. The rest speed increase in 1-5% is explained by absence of WOW64 layer which translates API calls between 32-bit applications and a 64-bit operation system.

If your program doesn't operate with large data sizes (more than 2GB) and the speed of its operation is not crucial, porting on a 64-bit system is not so urgent in the near future.

By the way, even simple 32-bit applications can get advantage being launched in 64-bit environment. Perhaps you know that a program built with `/LARGEADDRESSAWARE:YES` key can allocate up to 3 GB of memory if the 32-bit Windows is launched with `/3gb` key. This very 32-bit program launched on a 64-bit system can allocate nearly 4 GB of memory (in practice about 3.5 GB).

2.3. Development of libraries

If you develop libraries, components or other elements with the help of which third-party developers create their software you should act quickly while creating the 64-bit version of your product. Otherwise, your clients interested in release of 64-bit versions will have to search for alternative solutions. For example, some developers of software-hardware security responded with a large delay to appearance of 64-bit programs and that made some clients search for other tools to protect their programs.

An additional advantage of releasing the 64-bit version of a library is that you can sell it as a separate product. Thus, your clients wishing to create both 32-bit and 64-bit applications will have to buy 2 different licenses. For example, this policy is used by Spatial Corporation when selling Spatial ACIS

library.

2.4. Dependence of your product on third-party libraries

Before you plan your work on creation of the 64-bit version of your product, find out if there are 64-bit versions of libraries and components used in it. Besides, learn about the pricing policy concerning the 64-bit version of a library. If there is no support provided, search for alternative solutions supporting 64-bit systems beforehand.

2.5. Using 16-bit applications

If your solutions still use 16-bit units it is high time you got rid of them. 16-bit applications in 64-bit Windows versions are not supported.

We should explain one thing here concerning using 16-bit installers. They are still used to install some 32-bit applications. There is a special mechanism which replaces some of the most popular 16-bit installers with their newer versions. It can cause a false opinion that 16-bit programs still operate in 64-bit environment. Remember: it is not so.

2.6. Assembler code

Don't forget that using a large size of Assembler code can significantly increase the cost of creating the 64-bit version of an application.

Having thought all the listed factors over and weighed all pros and cons, decide if you need to port your project on 64-bit systems. If yes, we go further.

3. The third step. Toolkit

If you have decided to develop the 64-bit version of your product and are ready to spend time on it, it is still not enough to guarantee success. The point is that you must possess the entire necessary toolkit and here you can face some difficulties.

Absence of a 64-bit compiler can be the simplest but the most insuperable problem. The article is being written in 2009 but there is still no 64-bit C++ Builder compiler by Codegear [7]. Its release is expected by the end of this year. It is impossible to avoid this problem, if only to rewrite the whole project using, for example, Visual Studio. But if everything is clear about absence of a 64-bit compiler, other similar problems can appear to be less transparent and occur only at the stage of porting the project on a new architecture. That's why we would like to advise you to find out beforehand if there are all the necessary components you will need to implement the 64-bit version of your product. You may face unpleasant surprises.

Of course it is impossible to list everything you may need for a project here, but I will continue the list which will help you to orient yourself and perhaps to remember about other things necessary to implement your 64-bit project:

3.1. A 64-bit compiler

There is hardly more to say about the importance of having a 64-bit compiler. It simply must be.

If you are planning to develop 64-bit applications using the latest (by the moment the article is written) Visual Studio 2008 version, the following Table 2 will help you understand which of the Visual Studio editions you need.

	Microsoft Visual C++ Express Edition	Visual Studio 2008 Standard	Visual Studio 2008 Professional	Visual Studio 2008 Team System
32-bit x86 compiler	YES	YES	YES	YES
64-bit x64 compiler and cross-compiler		YES	YES	YES
64-bit Itanium compiler and cross-compiler				YES

Table 2. Abilities of different editions of Visual Studio 2008.

3.2. 64-bit computers under control of 64-bit operation system

Of course you can use virtual machines for launching 64-bit applications on 32-bit computers but it is too inconvenient and won't provide the necessary level of tests. It is desirable that the machines have not less than 4-8 GB of main memory.

3.3. 64-bit versions of all the used libraries

If libraries are presented in source codes, there must be a 64-bit configuration of the project. It can be a thankless and difficult task to update the library for a 64-bit system on your own, and the result can be unreliable and contain errors. Besides, you can violate license agreements by these actions. If you use libraries in the form of binary units you should also find out if there are 64-bit units. You cannot use 32-bit DLL inside a 64-bit application. You can create a special tie through COM but it will be a separate large and difficult task [8]. Also keep in mind that you may need to spend some extra money to purchase the 64-bit version of the library.

3.4. Absence of embedded Assembler code

Visual C++ doesn't support a 64-bit inline assembler. You must either use an external 64-bit assembler (for example, MASM) or possess an implementation with the same functionality in C/C++ [9].

3.5. Testing methodology update

It means considerable remaking of the testing methodology, update of unit-tests and using new tools. We will speak about it in more detail further, but don't forget to take it into account at the stage of evaluating time costs on migration of an application on a new system [10].

3.6. New data for testing

If you are developing resource-intensive applications using a large size of main memory, you need to provide replenishment of the testing input data base. At load testing of 64-bit applications it is desirable to excess the limits of 4 GB of the used memory. Many errors can occur only in these conditions.

3.7. 64-bit security systems

The security system being used must provide full support of 64-bit systems. For example, Aladdin

company has released 64-bit drivers for support of hardware Hasp keys rather quickly. But for a long time there has been no system of automatic protection of 64-bit binary files (Hasp Envelop program). Thus, the security mechanism had to be implemented manually inside the program code and that was one more difficult task demanding professionalism and time. Don't forget about such things relating to security, update system etc.

3.8. Installer

You need a new installer able to fully install 64-bit applications. We would like to warn you about one very typical mistake. It is creation of 64-bit installers for installing 32/64-bit program products. Preparing the 64-bit version of an application developers often want to make "64-bit mode" in it absolute and create a 64-bit installer forgetting that those who use a 32-bit operation system won't simply be able to launch such an installation package. Pay attention that it is not the 32-bit application included into the distribution kit together with the 64-bit one, but the installer itself. For if the distribution kit is a 64-bit application, of course it won't operate on a 32-bit operation system. What is the most unpleasant is that a user won't be able to guess why it happens. He will simply see the installation package which cannot be launched.

4. The fourth step. Setting of a project in Visual Studio 2005/2008

Creation of the 64-bit configuration of a project in Visual Studio 2005/2008 looks rather simple. Difficulties will begin at the stage of building a new configuration and searching errors in it. To create the 64-bit configuration itself you just need to perform the following 4 steps:

Launch the configuration manager as shown in Figure 1:

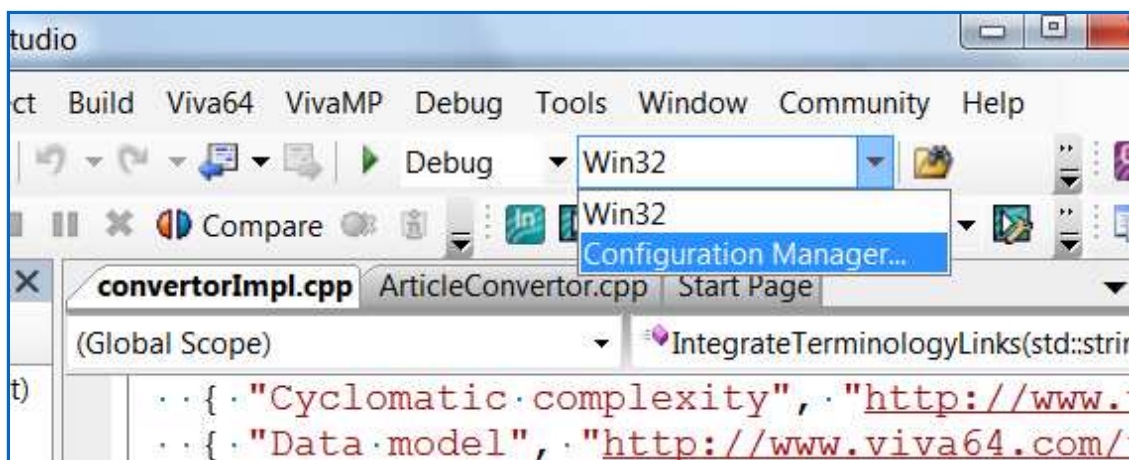


Figure 1. Launch of the configuration manager.

In the configuration manager choose support of the new platform (Figure 2):

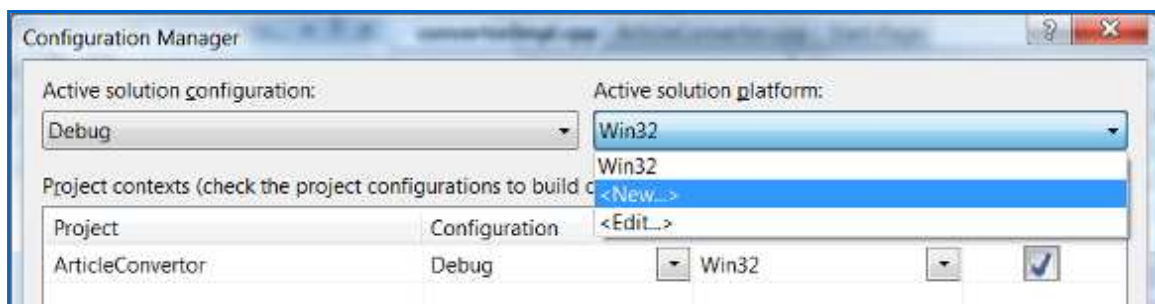


Figure 2. Creation of a new configuration.

Choose the 64-bit platform (x64) and as a basis - settings from the 32-bit version (Figure 3). Those settings

which influence the building mode will be corrected by Visual Studio automatically.

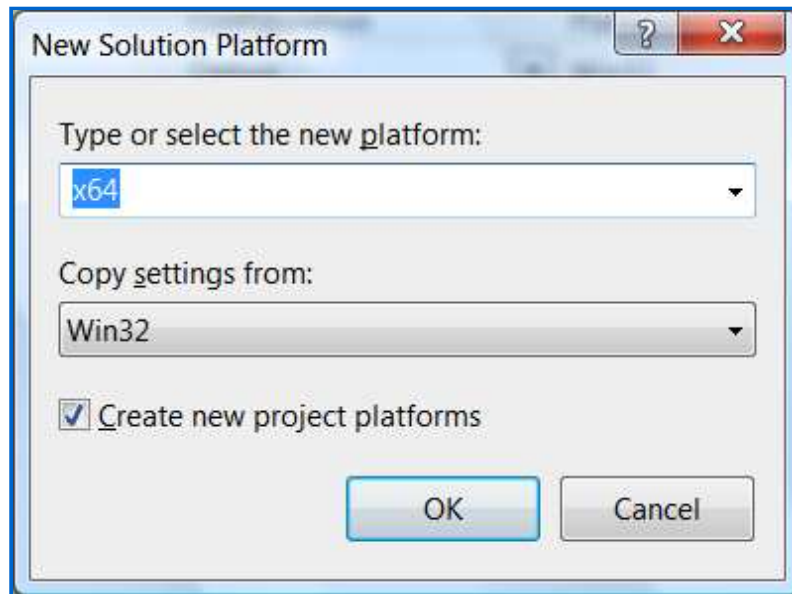


Figure 3. Choose x64 as a platform and use Win32 configuration as a basis.

Addition of a new configuration is complete and now you can choose the 64-bit configuration version and begin compiling a 64-bit application. Choosing the 64-bit configuration for building is shown in Figure 4.

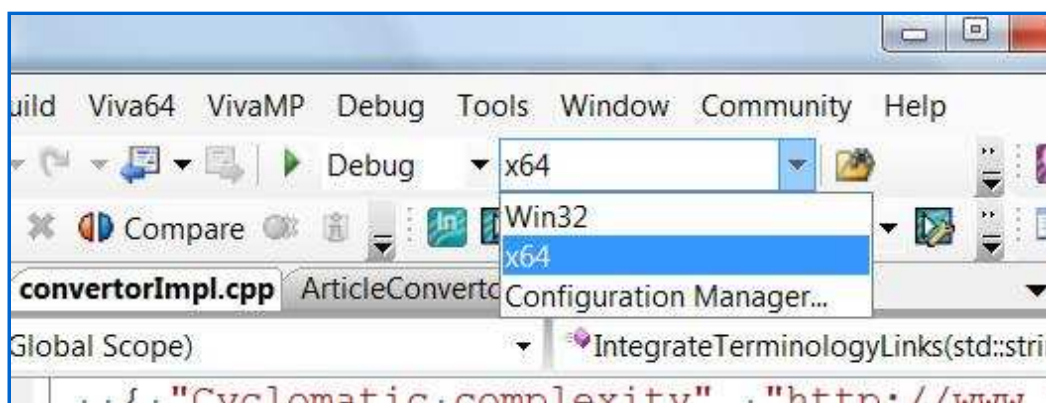


Figure 4. Now both 32-bit and 64-bit configurations are available.

If you are lucky you won't need to additionally set a 64-bit project. But it greatly depends on the project, its complexity and the number of libraries used. The only thing you should change at once is the stack's size. In case if the stack's size in your project is set by default, that is 1 MB, you should define it as 2 MB for the 64-bit version. It is not necessary but it is better to insure yourself beforehand. If you use the size different from that by default, there is sense in increasing it twice for the 64-bit version. To do this find and change Stack Reserve Size and Stack Commit Size parameters in the project's settings.

5. The fifth step. Compilation of an application

Here we should tell you about typical problems occurring at the stage of compiling the 64-bit configuration, discuss what problems occur in third-party libraries, tell you that in the code relating to WinAPI functions the compiler won't permit placing of a pointer into LONG type and you will have to update your code and use LONG_PTR type. And there is a lot else to say. Unfortunately, there are so many problems and errors are so various that we cannot describe them all in one article and even book. You will have to look through all the errors the compiler will show you and all the new warnings which haven't been there before by yourself and in each particular case to find out how to update the code.

The following list of links to resources devoted to developing 64-bit applications can partly help you: <http://www.viva64.com/links/64-bit-development/>. The list is constantly enlarged and the author will be glad if readers send him links to resources which are, in their opinion, worthy attention.

Let's describe here only types which can be of interest for developers when porting applications. These types are shown in Table 3. Most recompilation errors will relate to using these very types.

Type	Type's size on x32 / x64 platform	Note
int	32 / 32	Basic type. On 64-bit systems remains 32-bit.
long	32 / 32	Basic type. On 64-bit Windows systems remains 32-bit. Keep in mind that in 64-bit Linux systems this type was extended to 64-bit. Don't forget about it if you develop code which should be compiled for Windows and Linux systems.
size_t	32 / 64	Basic unsigned type. The type's size is chosen in such a way that you could write the maximum size of a theoretically possible array into it. You can safely put a pointer into size_t type (except for pointers to class functions, but this is a special case).
ptrdiff_t	32 / 64	Similar to size_t type but this is a signed type. The result of the expression where one pointer is subtracted from the other (ptr1-ptr2) will have ptrdiff_t type.
Pointer	32 / 64	The size of the pointer directly depends on the platform's size. Be careful while converting pointers to other types.
__int64	64 / 64	Signed 64-bit type.
DWORD	32 / 32	32-bit unsigned type. In WinDef.h is defined as:typedef unsigned long DWORD;
DWORDLONG	64 / 64	64-bit unsigned type. In WinNT.h is defined as:typedef ULONGLONG DWORDLONG;
DWORD_PTR	32 / 64	Unsigned type in which a pointer can be placed. In BaseTsd.h is defined as:typedef ULONG_PTR DWORD_PTR;
DWORD32	32 / 32	32-bit unsigned type. In BaseTsd.h is defined as:typedef unsigned int DWORD32;
DWORD64	64 / 64	64-bit unsigned type. In BaseTsd.h is defined as:typedef unsigned __int64 DWORD64;
HALF_PTR	16 / 32	A half of a pointer. In Basetsd.h is defined as:#ifdef _WIN64 typedef int HALF_PTR;#else typedef short HALF_PTR;#endif
INT_PTR	32 / 64	Signed type in which a pointer can be placed. In BaseTsd.h is defined as:#if defined(_WIN64) typedef __int64 INT_PTR; #else typedef int INT_PTR;#endif
LONG	32 / 32	Signed type which remained 32-bit. That's why in many cases LONG_PTR now should be used. In WinNT.h is defined as:typedef long LONG;
		Signed type in which a pointer can be placed. In BaseTsd.h is defined

LONG_PTR	32 / 64	as:#if defined(_WIN64) typedef __int64 LONG_PTR; #else typedef long LONG_PTR;#endif
LPARAM	32 / 64	Parameter for sending messages. In WinNT.h is defined as:typedef LONG_PTR LPARAM;
SIZE_T	32 / 64	Analog of size_t type. In BaseTsd.h is defined as:typedef ULONG_PTR SIZE_T;
SSIZE_T	32 / 64	Analog of ptrdiff_t type. In BaseTsd.h is defined as:typedef LONG_PTR SSIZE_T;
ULONG_PTR	32 / 64	Unsigned type in which a pointer can be placed. In BaseTsd.h is defined as:#if defined(_WIN64) typedef unsigned __int64 ULONG_PTR;#else typedef unsigned long ULONG_PTR;#endif
WORD	16 / 16	Unsigned 16-bit type. In WinDef.h is defined as:typedef unsigned short WORD;
WPARAM	32 / 64	Parameter for sending messages. In WinDef.h is defined as:typedef UINT_PTR WPARAM;

Table 3. Types to be noted while porting 32-bit programs on 64-bit Windows systems.

6. Diagnosis of hidden errors

If you think that after correcting all the compilation errors you will get a long-expected 64-bit application we have to disappoint you. The most difficult is just ahead. At the stage of compilation you will correct the most explicit errors which the compiler had managed to detect and which mostly relate to impossibility of implicit type conversion. But this is only a small part of the problem. Most errors are hidden. From the viewpoint of the abstract C++ language these errors look safe and are disguised by explicit type conversions. The number of such errors is much larger than the number of errors detected at the stage of compilation.

You shouldn't set your hopes on [/Wp64](#) key. This key is often presented as a wonderful means of searching 64-bit errors. In reality /Wp64 key just allows you to get some warning messages concerning incorrectness of some code sections in 64-bit mode while compiling 32-bit code. While compiling 64-bit code these warnings will be shown anyway. And that's why /Wp64 key is ignored when compiling a 64-bit application. And surely this key won't help in search of hidden errors [\[11\]](#).

Let's consider several examples of hidden errors.

6.1. Explicit type conversion

The simplest but none the easiest for detection error class relates to [explicit type conversions](#) when significant bits are cut. A popular example is conversion of pointers to 32-bit types when transferring them into functions such as SendMessage:

```
MyObj* pObj = ...
::SendMessage(hwnd, msg, (WORD)x, (DWORD)pObj);
```

Here the explicit type conversion is used to turn a pointer into a numeric type. For a 32-bit architecture this example is correct as the last parameter of SendMessage function has LPARAM type which coincides with DWORD on a 32-bit architecture. For a 64-bit architecture DWORD is incorrect and must be replaced with LPARAM. LPARAM type has sizes of 32 or 64 bits depending on the architecture.

This is a simple case but type conversion often looks more complicated and it is impossible to detect it using the compiler's warnings or search through the program text. Explicit type conversions suppress the compiler's diagnosis as they are intended for this very purpose - to tell the compiler that the type conversion is correct and the programmer is responsible for the code's safety. Explicit search won't help as well. Types can have non-standard names (defined by the programmer through typedef), and the number of methods to perform explicit type conversion is also large. To safely diagnose such errors you must use only a special toolkit such as [Viva64](#) or [PC-Lint](#) analyzers.

6.2. Implicit type conversion

The next example relates to [implicit type conversion](#) when significant bits are also lost. `fread` function's code performs reading from the file but it is incorrect when trying to read more than 2 GB on a 64-bit system.

```
size_t __fread(void * __restrict buf,
size_t size, size_t count, FILE * __restrict fp);

size_t
fread(void * __restrict buf, size_t size, size_t count,
FILE * __restrict fp)
{
    int ret;
    FLOCKFILE(fp);
    ret = __fread(buf, size, count, fp);
    FUNLOCKFILE(fp);
    return (ret);
}
```

`__fread` function returns `size_t` type but `int` type is used to store the number of the bytes read. As a result at large sizes of read data the function can return a false number of bytes.

You can say that it is an illiterate code for beginners, that the compiler will announce about this type conversion and that this code is actually easy to find and to correct. This is in theory. And in practice everything may be quite different in cases of large projects. This example is taken from FreeBSD source code. The error was corrected only in December 2008! Note that the first (experimental) 64-bit version of FreeBSD was released in June 2003.

This is the source code before it has been corrected:

<http://www.freebsd.org/cgi/cvsweb.cgi/src/lib/libc/stdio/fread.c?rev=1.14>

And this is the corrected variant (December 2008):

<http://www.freebsd.org/cgi/cvsweb.cgi/src/lib/libc/stdio/fread.c?rev=1.15>

6.3. Bits and shifts

It is easy to make an error in the code while working with separate bits. The following error type relates to shift operations. Here is an example:

```
ptrdiff_t SetBitN(ptrdiff_t value, unsigned bitNum) {
    ptrdiff_t mask = 1 << bitNum;
    return value | mask;
}
```

This code works well on a 32-bit architecture and allows you to set bits with numbers 0 to 31 to unity.

After porting the program on a 64-bit platform you will need to set bits 0 to 63. But this code will never set bits 32-63. Pay attention that "1" has int type and when a shift at 32 positions occurs an overflow will take place as shown in Figure 5. Whether we will get 0 (Figure 5-B) or 1 (Figure 5-C), as a result, depends on the compiler's implementation.

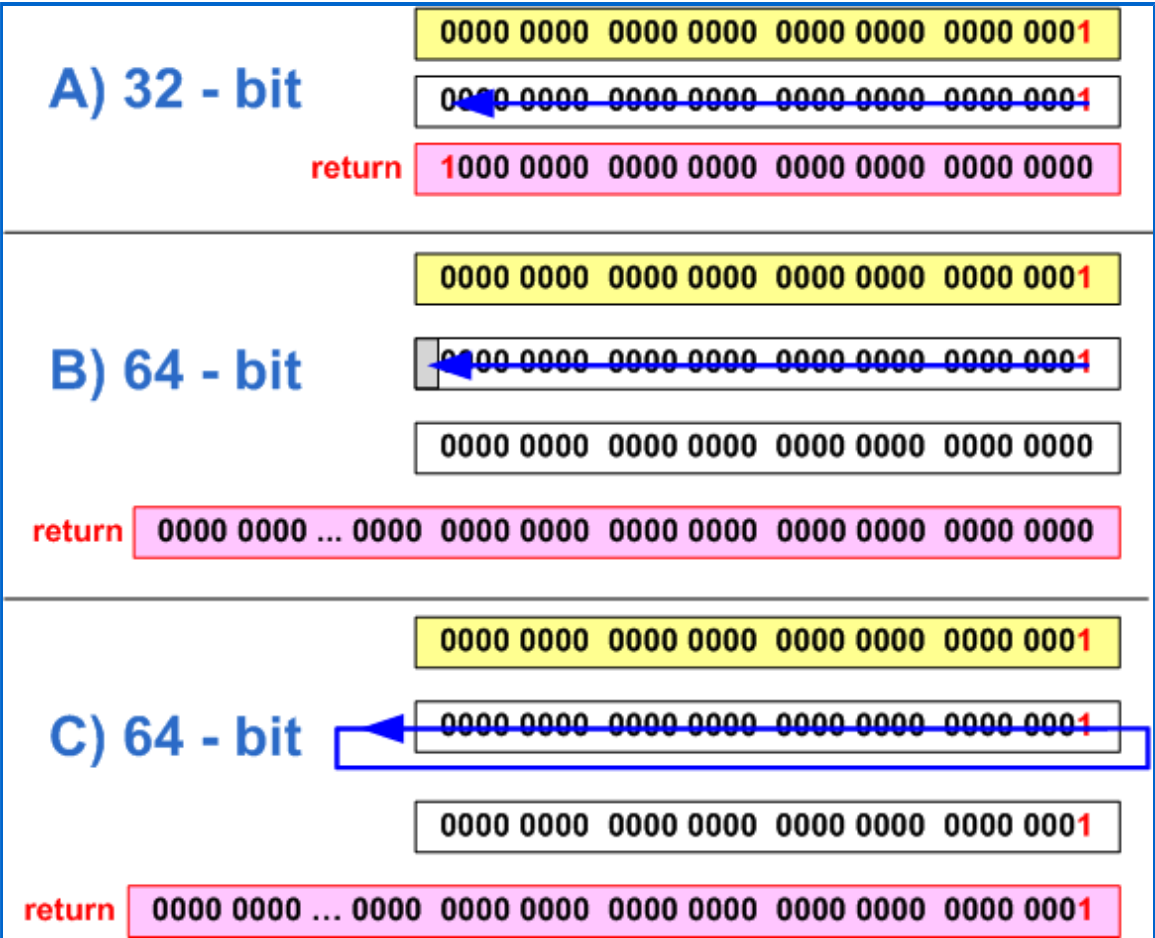


Figure 5. A - Correct setting of the 32th bit in 32-bit code; B,C - error of setting of the 32th bit on a 64-bit system (two ways of behavior)

To correct the code we need to make "1" constant of the same type as mask variable:

```
ptrdiff_t mask = ptrdiff_t(1) << bitNum;
```

Also pay attention that the incorrect code leads to one more error. When setting 31 bits on a 64-bit system the result of the function will be the value 0xffffffff80000000 (see Figure 6). The result of 1 << 31 expression is the negative number -2147483648. In a 64-bit integer variable this number is presented as 0xffffffff80000000.

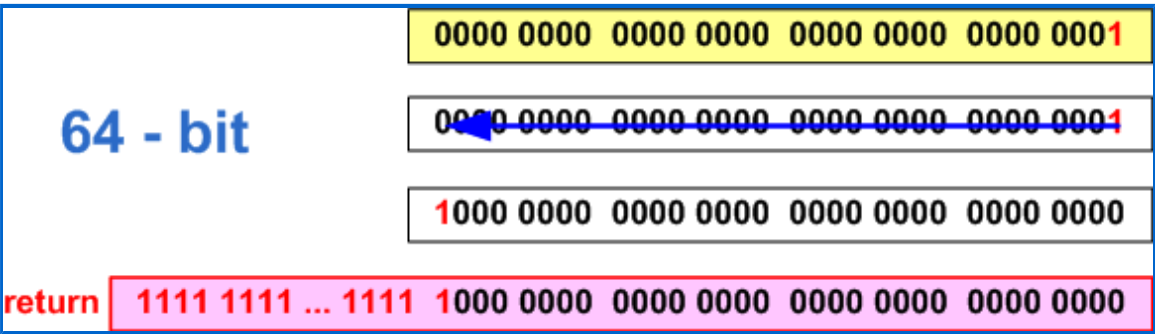


Figure 6. Error of setting of the 31th bit on a 64-bit system

6.4. Magic numbers

Magic constants, i.e. numbers with the help of which the size of this or that type is defined, can cause a lot of troubles. A proper decision is to use `sizeof()` operators for these purposes, but in a large program an old code section can still be hidden where, as programmers believe, the pointer's size is 4 bytes and in `size_t` it is always 32 bits. Usually such errors look as follows:

```
size_t ArraySize = N * 4;
size_t *Array = (size_t *)malloc(ArraySize);
```

Figure 4 shows the basic numbers with which you should work with caution while migrating on a 64-bit platform.

Value:	Can be used as:
4	Number of bytes in type.
32	Number of bits in type.
0x7fffffff	Max value of signed variable. Mask for higher bit zero setting.
0x80000000	Min value of signed variable. Mask for higher bit selecting.
0xffffffff	Max value of unsigned variable. Alternative representation -1 as error indicator.

Table 4. Basic magic values which are dangerous while porting applications from a 32-bit platform to a 64-bit one.

6.5. Errors relating to using 32-bit variables as indexes

In programs processing large data sizes errors relating to indexing large arrays or eternal loops may occur. The following example contains 2 errors:

```
const size_t size = ...;
char *array = ...;
char *end = array + size;
for (unsigned i = 0; i != size; ++i)
{
    const int one = 1;
    end[-i - one] = 0;
}
```

The first error consists in that if the size of the data being processed exceeds 4 GB (0xFFFFFFFF) an eternal loop may occur as 'i' variable has 'unsigned' type and will never reach 0xFFFFFFFF value. I write deliberately that it *can* occur but not necessarily. It depends on what code the compiler will build. For example, in debug mode the eternal loop will be present and in release-code there will be no loop as the compiler will decide to optimize the code using a 64-bit register for the counter and the loop will be correct. All this adds much confusion and the code which worked yesterday can fail to work today.

The second error relates to parsing the array from beginning to end for what negative indexes' values are used. This code will operate well in 32-bit mode but when executed on a 64-bit computer access outside

the array's limits will occur at the first iteration of the loop, and there will be a program crash. Let's study the reason of such a behavior.

According to C++ rules "-i - one" expression on a 32-bit system will be calculated as follows: (at the first step $i = 0$):

"-i" expression has unsigned type and has $0x00000000u$ value.

'one' variable will be extended from 'int' type to unsigned type and will equal $0x00000001u$. Note: int type is extended (according to C++ standard) up to 'unsigned' type if it participates in an operation where the second argument has unsigned type.

A subtraction operation takes place in which two values of unsigned type participate and the result of the operation equals $0x00000000u - 0x00000001u = 0xFFFFFFFFu$. Note that the result will have unsigned type.

On a 32-bit system access to the array by the index $0xFFFFFFFFu$ is the same as using -1 index. That is $\text{end}[0xFFFFFFFFu]$ is an analog of $\text{end}[-1]$. As a result the array's items will be processed correctly.

In a 64-bit system the situation will be quite different concerning the last point. Unsigned type will be extended to signed `ptrdiff_t` type and the array's index will equal $0x00000000FFFFFFFFi64$. As a result an overflow will occur.

To correct the code you should use `ptrdiff_t` and `size_t` types.

6.6. Errors relating to change of the types of the used functions

There are errors which are nobody's fault but they are still errors. Imagine that long long ago in a faraway galaxy (in Visual Studio 6.0) a project was developed which contained `CSampleApp` class - a successor of `CWinApp`. In the basic class there is a virtual function `WinHelp`. The successor overlaps this function and performs all the necessary actions. This process is shown in Figure 7.

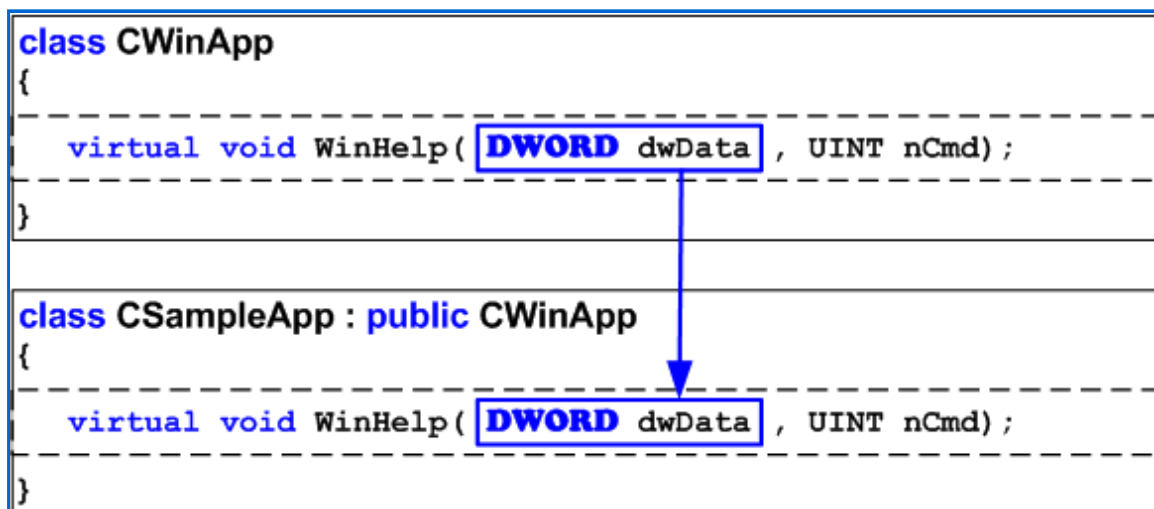


Figure 7. Efficient correct code created in Visual Studio 6.0.

After that the project is ported on Visual Studio 2005 where the prototype of `WinHelp` function has changed but nobody will notice it because in 32-bit mode `DWORD` and `DWORD_PTR` types coincide and the program continues operating correctly (Figure 8).

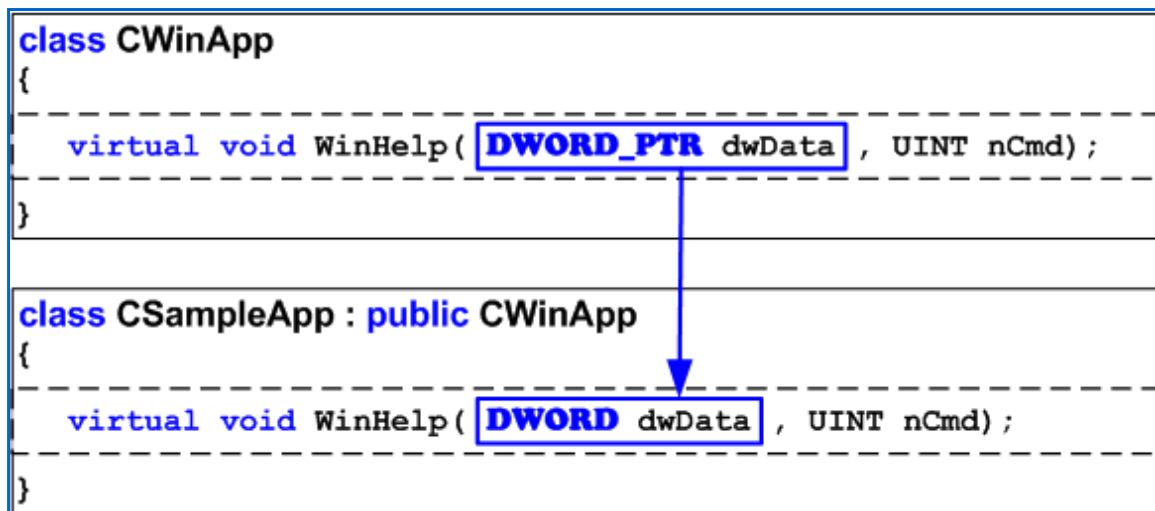


Figure 8. Incorrect but efficient 32-bit code.

The error is waiting to occur on a 64-bit system where the sizes of `DWORD` and `DWORD_PTR` types are different (Figure 9). In 64-bit mode classes appear to contain two DIFFERENT functions `WinHelp` and that is of course incorrect. Keep in mind that such traps may hide not only in MFC where some functions have changed their arguments' types but in the code of your applications and third-party libraries as well.

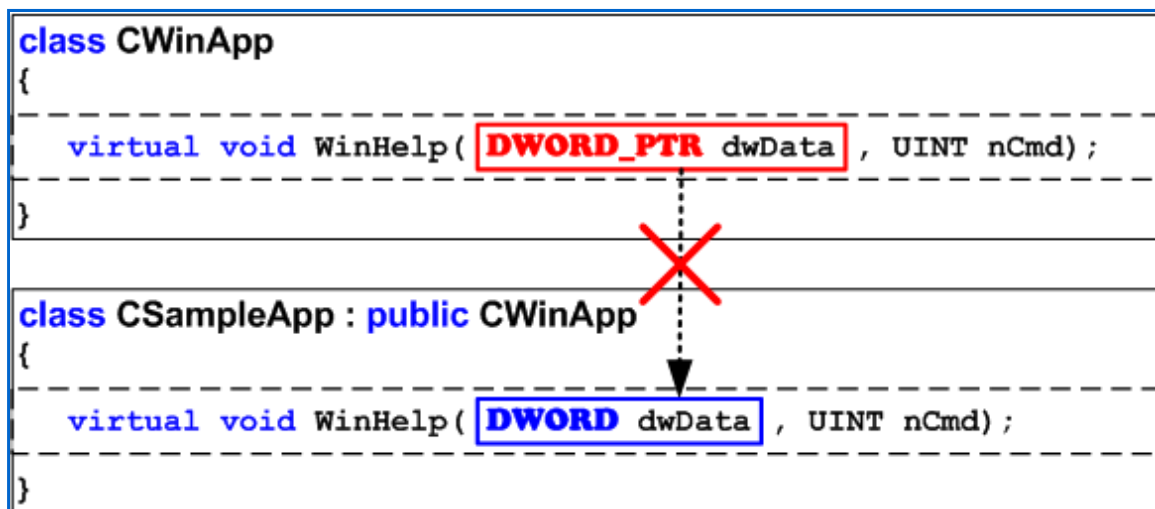


Figure 9. The error occurs in 64-bit code

6.7. Diagnosis of hidden errors

There are a lot of examples of such 64-bit errors. Those who are interested in this topic and would like to know more about these errors see the article "20 issues of porting C++ code on the 64-bit platform" [12].

As you see the stage of searching hidden errors is a nontrivial task, and besides, many of them will occur irregularly and only at large input data. Static code analyzers are good for diagnosing such errors as they can check the whole code of an application independently from the input data and the frequency of its sections' execution in real conditions. There is sense in using static analysis both at the stage of porting an application on 64-bit platforms to find most errors at the very beginning and in further development of 64-bit solutions. Static analysis will warn and teach a programmer to better understand the peculiarities of errors relating to a 64-bit architecture and to write more efficient code. The author of the article is a developer of one of such specialized code analyzers named Viva64 [13]. To learn more about the tool and to download a demo version visit the site of [OOO "Program Verification Systems"](#) company.

For justice' sake we should say that [Gimpel PC-Lint](#) and [Parasoft C++test](#) code analyzers have sets of rules for diagnosing 64-bit errors. But, firstly, these are general-purpose analyzers and the rules of diagnosing

64-bit errors are incomplete. Secondly, they are intended mostly for [LP64](#) data model used in the family of Linux operation system and that's why they are not so useful for Windows programs where [LLP64](#) data model is used [14].

7. The seventh step. Update of the testing process

The step of searching errors in program code described in the previous section is necessary but insufficient. None of the methods, including static code analysis, can guarantee detection of all the errors, and the best result can be achieved only when combining different methods.

If your 64-bit program processes a larger data size than the 32-bit version, you need to extend tests to include processing data with the size more than 4 GB. This is the border beyond which many 64-bit errors begin to occur. Such tests may take much more time and you must be prepared for it. Usually tests are written in such a way that each test could process a small number of items and thus make it possible to perform all the internal unit-tests in several minutes while automatic tests (for example, using [AutomatedQA TestComplete](#)) could be performed in several hours. It is nearly absolutely certain that the sorting function sorting 100 items will behave correctly at 100000 items on a 32-bit system. But the same function can fail on a 64-bit system while trying to process 5 billion items. The speed of executing a unit-test can fall in million times. Don't forget about the cost of adapting tests while mastering 64-bit systems. A good solution is to divide unit-tests into quick (working with small memory sizes) and slow ones processing gigabytes and executed, for example, in the nighttime. Automated testing of resource-intensive 64-bit programs can be organized on the basis of distributed calculations.

There is one more unpleasant thing. You will hardly succeed in using tools like [BoundsChecker](#) for searching errors in resource-intensive 64-bit programs consuming large memory size. The reason is a great slowdown of the programs being tested what makes this approach very inconvenient. In the mode of diagnosing all the errors relating to memory operation, [Parallel Inspector](#) tool included into [Intel Parallel Studio](#) will slow down execution of an application in 100 times on the average (Figure 10). It is very likely that you will have to leave the algorithm being tested for the night to see the results only the next day while normally this algorithm operates just 10 minutes. And still I'm sure that Parallel Inspector is one of the most useful and convenient tools when working in the mode of searching memory-operation errors. You just must be ready to change the practice of error diagnosing and keep it in mind when planning to master 64-bit systems.

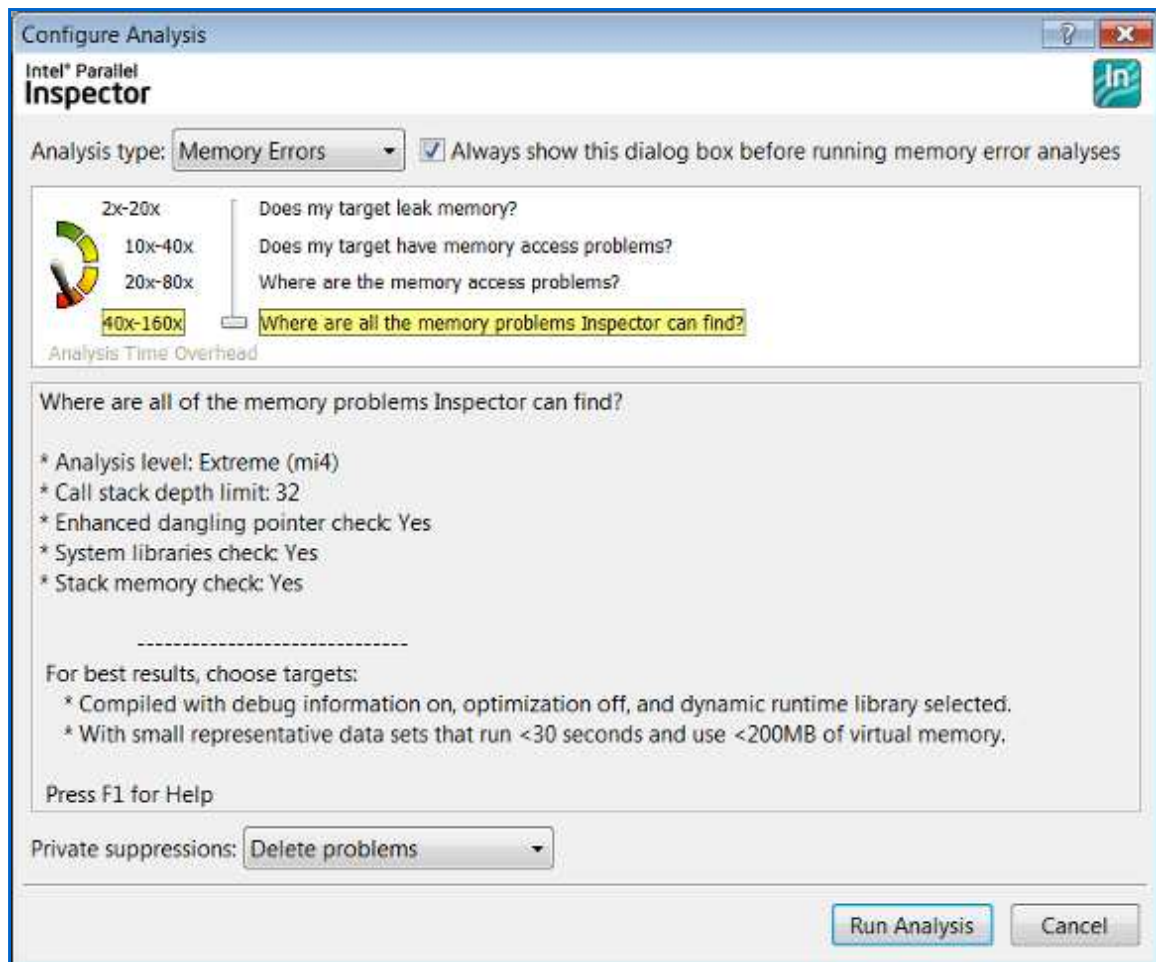


Figure 10. The settings window of Parallel Inspector program before launching an application.

And the last thing. Don't forget to add tests checking compatibility of data formats between the 32-bit and 64-bit versions. Data compatibility is often violated during migration because of writing of such types as `size_t` or `long` (in Linux systems) into files.

References

1. Wikipedia. 64-bit. <http://www.viva64.com/go.php?url=203>
2. Wikipedia. AMD64. <http://www.viva64.com/go.php?url=220>
3. Sverre Jarp. IA-64 architecture. A Detailed Tutorial. <http://www.viva64.com/go.php?url=222>
4. Wikipedia. Itanium. <http://www.viva64.com/go.php?url=221>
5. Andrey Karpov. The forgotten problems of 64-bit programs development <http://www.viva64.com/art-1-2-16511733.html>
6. Wikipedia. WOW64. <http://www.viva64.com/go.php?url=207>
7. Nick Hodges. The Future of the Delphi Compiler. <http://www.viva64.com/go.php?url=208>
8. Mike Becker. Accessing 32-bit DLLs from 64-bit code. <http://www.viva64.com/go.php?url=209>
9. Eric Palmer. How to use all of CPUID for x64 platforms under Microsoft Visual Studio .NET 2005. <http://www.viva64.com/go.php?url=210>
10. Andrey Karpov, Evgeniy Ryzhkov. Traps detection during migration of C and C++ code to 64-bit Windows. <http://www.viva64.com/art-1-2-2140958669.html>
11. Andrey Karpov. 64 bits, /Wp64, Visual Studio 2008, Viva64 and all the rest... <http://www.viva64.com/art-1-2-621693540.html>
12. Andrey Karpov, Evgeniy Ryzhkov. 20 issues of porting C++ code on the 64-bit platform. <http://www.viva64.com/art-1-2-599168895.html>
13. Evgeniy Ryzhkov. Viva64: what is it and who is it for? <http://www.viva64.com/art-1-2-903037923.html>

14. Andrey Karpov. Comparison of analyzers' diagnostic possibilities at checking 64-bit code.
<http://www.viva64.com/art-1-2-914146540.html>